

## Datenstrukturen

---

- Arrays
- Collections
- Lists
- Sets
- Maps

## Arrays

---

- **Arrays:** rudimentärste Art, mehrere gleichartige Objekte in Java zu speichern
  - Elemente werden sequenziell hintereinander in den Hauptspeicher geschrieben
  - Zugriff auf ein Element durch Angabe der Index
  - Index eines Elementes: Position innerhalb des für den Array reservierten Speicherbereichs
  - Index beginnt stets mit 0
  - Index des letzten Element eines Arrays mit n Elementen ist stets [n-1]

# Arrays

## ➤ Deklaration eines **Arrays**:

- Datentyp gefolgt von einer geöffneten und einer geschlossenen eckigen Klammer und dem Bezeichner

```
private Kunde[] kunden;  
...
```

Datentyp der Elemente

Name des Arrays

- Bei der Deklaration wird die Größe des Arrays nicht angegeben. Es wird daher zu diesem Zeitpunkt noch kein Speicherplatz für den Array reserviert

# Arrays

## ➤ Bei der **Instanziierung** des Arrays wird Speicherplatz reserviert

- Instanziierung erfolgt mit dem Schlüsselwort **new**
- In den eckigen Klammern ist die gewünschte Kapazität anzugeben
- Bei der Instanziierung ist zu beachten, dass die Elemente mit dem Standardwert des jeweiligen Datentyps vorbelegt werden:
  - Ein **int**-Array wird mit lauter Nullen gefüllt
  - Ein **boolean**-Array mit **false**-Werten
  - Arrays für komplexe Datentypen (z. B. Strings und eigene Klassen) mit **null**-Werten

# Arrays

## Deklaration und Instanziierung eines Arrays

```
public class Kundenverwaltung {  
    private Kunde[] kunden;      ← Deklaration des Arrays  
    ...  
    public Kundenverwaltung(){  
        kunden = new Kunde[42]; ← Instanziierung mit Kapazität 42  
                                   Alle Elemente werden mit Datentyp-  
                                   spezifischen Standard-Werten belegt  
        System.out.println(kunden[0]); ← null (Standard-Wert)  
        System.out.println(kunden[41]);  
        System.out.println(kunden[42]); ← ArrayIndexOutOfBoundsException  
    }  
}
```

22.12.2016

Monika Tepfenhart

5

# Arrays

- Für eine andere Vorbelegung (keine Standardwerte) kann die Instanziierung auch mit einer Initialisierung einhergehen
  - In geschweiften Klammern wird eine Komma-getrennte Liste von Wertausprägungen angegeben
  - Durch Angabe der Initialwerte wird implizit die Kapazität des Arrays festgelegt - die Angabe der Kapazität fällt weg.
  - Die Initialisierung kann nur zusammen mit der Instanziierung erfolgen und nicht getrennt in einer späteren Anweisung

22.12.2016

Monika Tepfenhart

6

## Arrays

### Instanziierung eines Arrays mit Initialisierung

```
public class Kundenverwaltung {  
    private Kunde[] kunden;  
    ...  
    public Kundenverwaltung(){  
        kunden = new Kunde[] {new Kunde("Ulf", "Koll"),  
                                new Kunde("Ilse", "Stahl"),  
                                new Kunde("Rita", "Kafka")};  
        System.out.println(kunden[0]);  
        System.out.println(kunden[1]);  
        System.out.println(kunden[2]);  
    }  
}
```

Kapazität wird implizit durch die Initialisierung vorgegeben

Instanziierung mit Initialisierung

Rechteckiges Ausschneiden

22.12.2016

Monika Tepfenhart

7

## Arrays

- Nach der Instanziierung kann die Kapazität eines Arrays nicht mehr verändert werden
- Überblick über die Kapazität mit Hilfe des Attribut `length` möglich
- Wichtig wenn eine separate Methode, in der die Größe des Arrays üblicherweise unbekannt ist, alle Elemente des Arrays verarbeiten möchte

22.12.2016

Monika Tepfenhart

8

# Arrays

## Attribut length am Beispiel der for-Schleife

```
public class Kundenverwaltung {  
    private Kunde[] kunden;  
    ...  
    public void aktualisiereAlleKunden(){  
        for (int index=0; i<kunden.length; index++)  
            if (kunden[index] != null)  
                kundenSpeicher.aktualisieren(kunden[index]);  
    }  
}
```

Schleife stoppt, sobald i kein gültiger Index mehr ist, d. h. i==kunden.length

Achtung: Prüfen, ob sich an der Index-Stelle tatsächlich ein Element befindet

pro Schleifendurchlauf wird ein Kunde aktualisiert

22.12.2016

Monika Tepfenhart

9

# Arrays

## ➤ In Java ist es möglich, Arrays zu verschachteln:

- Die Elemente eines Arrays sind dann ebenfalls Arrays
- Man spricht dann von mehrdimensionalen Arrays, da sich die Größe des Arrays bildlich gesehen nicht nur in eine Dimension ausdehnt, sondern in mindestens zwei
- Möglicher Anwendungsfall: ein Schachbrett-Array, das zu jeder Zeile jeweils ein Array mit den dazugehörigen Spielfeldern enthält

22.12.2016

Monika Tepfenhart

10

# Arrays

## „Normale“ und mehrdimensionale Arrays

← „normaler“, 1-dimensionaler Array

```
int[] vektor = new int[] {2, 4, 1};
```

```
int[][] matrix = new int[][] {{7, 3, 2},  
                               {9, 2, 7},  
                               {0, 3, 3},  
                               {1, 0, 6}};
```

2 Klammer-Paare  
=> 2 Dimensionen

← verschachtelter, 2-dimensionaler  
Array

# Arrays

## ➤ Vorteile:

- Deklaration und Verwendung unmittelbarer Bestandteil der Java-Syntax
- Daher nicht nötig Bibliotheken zu importieren
- Arrays können beliebige Typen enthalten: primitiven Datentypen, Strings und auch selbst programmierte Klassen

## Arrays

---

### ➤ Nachteile:

- Bei Arrays muss man sich selbst um die Kapazität kümmern - im Gegensatz zu Collections
- Array voll:
  - Es muss es zur Laufzeit mit einer größeren Kapazität neu initialisiert werden und alle Elemente müssen übertragen werden
- zu hohe Kapazität und folglich unnötigerweise ein viel zu großer Speicherbereich ebenfalls möglich

---

22.12.2016

Monika Tepfenhart

13

## Arrays

---

### ➤ Nachteile:

- Lücken in sortierten Arrays zu schließen ist mit großen Anstrengungen verbunden
  - fürs Aufrücken muss jedes Folgeelement bewegt werden
- Arrays haben eine begrenzte eingebaute Funktionalität:
  - zusätzlicher Programmieraufwand für die Form eines Stapels, einer Warteschlange oder einer Menge

---

22.12.2016

Monika Tepfenhart

14

## Collections

- Das **Collections - Framework** ist eine Menge an häufig benötigten Datenstrukturen und dazu passenden Such- und Sortialgorithmen
- Die Gemeinsamkeiten der Collections befinden sich im Interface Collection
- Bis auf Datenstrukturen zur Realisierung von Mengen wird dieses Interface (oder daraus abgeleitete Interfaces) von allen Klassen des Collections-Frameworks implementiert

22.12.2016

Monika Tepfenhart

15

## Collections

- **Hauptaufgaben:**
  - Daten effizient zu speichern und
  - effizienten Zugriff auf die Daten zu ermöglichen
- Beides sind konkurrierende Ziele, daher:
  - Wahl zwischen verschiedene Implementierungen, entweder:
    - sparsame Speicherung oder
    - schneller Zugriff begünstigen

22.12.2016

Monika Tepfenhart

16



# Collections

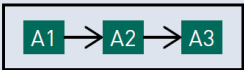
## Die wichtigsten Methoden der Schnittstelle Collection:

Platzhalter für den Typ der gespeicherten Objekte.  
Wird erst zur Laufzeit bzw. bei der Deklaration definiert („Generics“)

```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);           fügt Objekt hinzu (Typ erst zur Laufzeit bekannt, s. o.)  
    boolean remove(Object o);   entfernt ein Objekt  
    int size();                 liefert Anzahl der Objekte zurück (nicht die Kapazität!)  
    boolean isEmpty();          prüft, ob irgendwelche Objekte enthalten sind  
    boolean contains(Object o); prüft, ob ein bestimmtes Objekt enthalten ist  
    void clear();               entfernt alle Objekte der Collection  
    Iterator<E> iterator();     liefert den Iterator zurück (s. nächste Lektion)  
    Object[] toArray();         liefert die Collection als einfachen Array zurück  
}
```


# Collections

## Die wichtigsten Collections:

Programmieraufgabe	Besondere Eigenschaften
Artikel im Warenkorb verwalten	Elemente in sequenziell geordneter Reihenfolge
Skizze	Interfaces/Klassen im Collections-Framework
	Interface <code>java.util.List</code> Beispiele für Implementierungen: <ul style="list-style-type: none"><li>• <code>java.util.ArrayList</code> (als Array realisiert)</li><li>• <code>java.util.LinkedList</code> (als Verkettung von Referenzdatentypen realisiert)</li></ul>

# Collections

## Die wichtigsten Collections:

Programmieraufgabe	Besondere Eigenschaften
Verwaltung des Shop-Sortiments	Keine doppelten Elemente; Reihenfolge egal
Skizze	Interfaces/Klassen im Collections-Framework
	Interface <code>java.util.Set</code> Beispiele für Implementierungen: <ul style="list-style-type: none"><li>• <code>java.util.TreeSet</code> (als Baum realisiert)</li><li>• <code>java.util.HashSet</code> (als Hash-Tabelle)</li></ul>

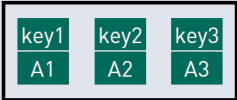
22.12.2016

Monika Tepfenhart

19

# Collections

## Die wichtigsten Collections:

Programmieraufgabe	Besondere Eigenschaften
Kundenverwaltung	Schneller Zugriff anhand der Kundennummer
Skizze	Interfaces/Klassen im Collections-Framework
	Interface <code>java.util.Map</code> Beispiele für Implementierungen: <ul style="list-style-type: none"><li>• <code>java.util.TreeMap</code> (als Baum realisiert)</li><li>• <code>java.util.HashMap</code> (als Hash-Tabelle)</li><li>• <code>java.util.LinkedHashMap</code> (Kombination aus Hash-Tabelle und verketteter Liste)</li></ul>

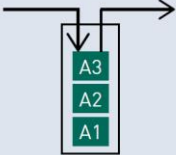
22.12.2016

Monika Tepfenhart

20

# Collections

## Die wichtigsten Collections:

Programmieraufgabe	Besondere Eigenschaften
„Undo“-Funktion im Bestellprozess	Die Bestellschritte des Benutzers sollen rückgängig gemacht werden können („Last in, First out“)
Skizze	Interfaces/Klassen im Collections-Framework
	Interface <code>java.util.Deque</code> Beispielimplementierung: <ul style="list-style-type: none"><li>• <code>java.util.ArrayDeque</code> (als Array realisiert)</li></ul> Interface <code>java.util.List</code> <ul style="list-style-type: none"><li>• <code>java.util.Stack</code></li></ul>

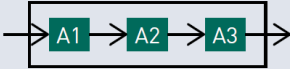
22.12.2016

Monika Tepfenhart

21

# Collections

## Die wichtigsten Collections:

Programmieraufgabe	Besondere Eigenschaften
Warteschlange für Bestellungen	Bearbeitung in der Reihenfolge des Eingangs („First in, First out“)
Skizze	Interfaces/Klassen im Collections-Framework
	Interface <code>java.util.Queue</code> Beispielimplementierung: <ul style="list-style-type: none"><li>• <code>java.util.LinkedList</code> (s. o.)</li></ul>

22.12.2016

Monika Tepfenhart

22

## Collections

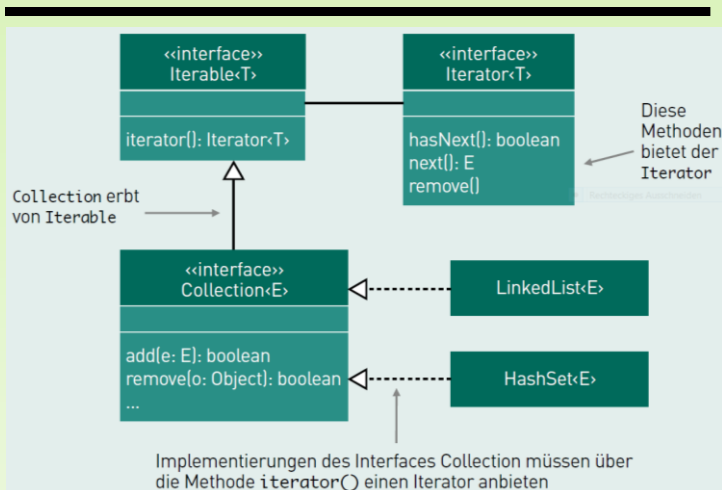
- **Iterator:** Sein Zweck ist es, über alle Elemente in einer beliebigen Collection zu iterieren.
- Iterieren bedeutet, alle Elemente, z. B. innerhalb einer Programmschleife, zu durchlaufen und bei Bedarf zu verarbeiten
- Im Collections-Framework hat jede Collection eine eigene Iterator-Implementierung
  - Sicherstellung mit dem Interface Iterable
  - besteht aus der Methode iterator() und wird vom Interface Collection erweitert

22.12.2016

Monika Tepfenhart

23

## Collections



22.12.2016

Monika Tepfenhart

24

## Collections

- Der Iterator besitzt drei Methoden:
  - `hasNext()`: prüft, ob es an der aktuellen Position des Iterators noch ein neues Element gibt
  - Übergabe mit der Methode `next()`. Der Iterator positioniert sich anschließend selbstständig auf das nächste Element
  - `remove()` kann das aktuelle Element, auf das sich der Iterator per `next()` zuvor positioniert hat, aus der Collection entfernt werden

22.12.2016

Monika Tepfenhart

25

## Collections

### Bereinigung von Datensätzen mit Hilfe eines Iterators:

Die Methode funktioniert mit jeder Collection, die Kunden enthält

```
public void bereinigeAlleKunden(Collection<Kunde> c) {  
    Kunde k = null;  
    for (Iterator<Kunde> it = c.iterator(); it.hasNext(); k = it.next())  
        if (!k.getGeschlecht().equals(""))  
            it.remove();  
}
```

Der Iterator ersetzt die Laufvariable

anstelle von `i++` (o. ä.) tritt `it.next()`

Schleifen-Bedingung erfüllt, solange noch Elemente folgen

Aufruf der Iterator-Methode zum Löschen des aktuellen Elements

22.12.2016

Monika Tepfenhart

26

## Collections

- Die Methode erwartet als Parameter:
  - beliebige Collection – d.h. alle Klassen, die das Interface Collection implementieren.
  - Einschränkung: Elemente der Collection Objekte der Klasse „Kunde“ oder einer ihrer Unterklassen
- Methode besteht aus einer for-Schleife:
  - anstelle einer Laufvariablen verwendet den Iterator
  - Bei der Initialisierung wird keine Laufvariable deklariert. Es wird eine Referenz auf den Iterator der Collection erzeugt.

22.12.2016

Monika Tepfenhart

27

## Collections

- Abbruchbedingung erfüllt, wenn:
  - Iterator findet keine weiteren Elemente, d.h. it.hasNext() wird zu false ausgewertet wird
  - Die Angabe einer Schrittweite (z. B. i++) ist bei einem Iterator überflüssig
  - Der Iterator auf den nächsten Kunden positioniert. Für die Weiterverarbeitung Um das Objekt im Rumpf der Schleife komfortabel weiterverarbeiten zu können, wird eine Referenz zwischengespeichert

22.12.2016

Monika Tepfenhart

28

## Collections

- Nützliches Werkzeug für das Arbeiten mit Collections ist die **erweiterte for-Schleife**:
  - vereinfacht Durchlaufen von beliebigen Klassen, die das Interface Iterable implementieren (auch Collections)
- Angaben im Kopf der Schleife:
  - Deklaration eines Stellvertreters für jedes Collection-Element und die zu verarbeitende Collection selbst
  - Argumente werden mit einem Doppelpunkt getrennt

22.12.2016

Monika Tepfenhart

29

## Collections

### Erweiterte for-Schleife als Ersatz für den Iterator

```
public void bereinigeAlleKunden(Collection<Kunde> c) {  
    for (Kunde k : c)  
        if (!k.getGeschlecht().equals(""))  
            c.remove(k);  
}
```

Stellvertreter-Objekt für alle Elemente

Collection, die durchlaufen werden soll

Rechteckiges Ausschneiden

Anweisungen im Rumpf der Schleife werden für jedes einzelne Element (repräsentiert durch k) ausgeführt

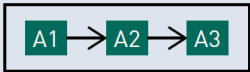
22.12.2016

Monika Tepfenhart

30

## Listen

**Liste:** Speichern beliebig vieler Elemente in sequenziell geordneter Reihenfolge

Skizze	Interfaces/Klassen im Collections-Framework
	Interface <code>java.util.List</code> Beispiele für Implementierungen: <ul style="list-style-type: none"><li>• <code>java.util.ArrayList</code> (als Array realisiert)</li><li>• <code>java.util.LinkedList</code> (als Verkettung von Referenzdatentypen realisiert)</li></ul>

22.12.2016

Monika Tepfenhart

31

## Listen

**Zusätzliche Methoden - außer der Collections-Interfaces**

- `void add(int index, E element)`
  - Fügt Element vom Typ E an Stelle index ein und verschiebt Rest nach Rechts
- `boolean addAll(int index, Collection<? extends E> c)`
  - Fügt ganze Collection an der Stelle index ein
  - Elemente der Collection können beliebigen Typs sein
  - Typ sollte zumindest in einer Vererbungsbeziehung mit dem Element-Typ der Ziel-Collection (E) stehen
  - Liefert true zurück, falls die Operation erfolgreich war

22.12.2016

Monika Tepfenhart

32



## Listen

- 
- `E remove(int index)`
    - Löscht Objekt an der Stelle `index`
    - Gibt Referenz auf das gelöschte Element zurück.
  - `set(int index, E element)`
    - Tauscht übergebenes Element mit Element an Position `index`
  - `subList(int fromIndex, int toIndex)`
    - Liefert Ausschnitt (von `fromIndex` bis exklusive `toIndex`)
  - **Achtung:** Liste ist Referenzdatentyp
    - Veränderungen an der Teil-Liste wirken sich aufs Original aus

---

22.12.2016

Monika Tepfenhart

33

## ArrayList

- 
- **ArrayList** speichert Elemente intern in einem Array
    - Zugriff auf Elemente äußerst schnell
    - Änderungen am `ArrayList` aufwendig (nachfolgende Listenelemente müssen ebenfalls verschoben werden)
    - Ausnahme: Element wird am Ende der Liste eingefügt bzw. gelöscht
    - Potentielle Probleme mit Größe des internen Arrays, siehe Nachteile für Arrays
    - `ArrayList` – komfortabler als herkömmliches Array

---

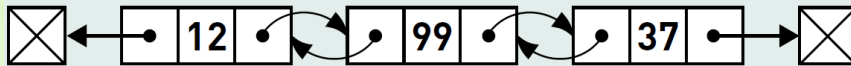
22.12.2016

Monika Tepfenhart

34

## LinkedList

- **LinkedList** ist als doppelt verkettete Liste realisiert
  - Liste besteht aus Objekten, die eine Referenz auf ihren Vorgänger und ihren Nachfolger halten
- Interne Speicherstruktur einer doppelt verketteten Liste



- LinkedList arbeitet ohne einen Index

## LinkedList

- Zugriff auf beliebiges Element:
  - Im schlimmsten Fall muss ganze Liste durchlaufen werden, um Referenz zu erhalten
    - Zugriff bei verketteter Liste langsamer, als bei der ArrayList
  - Deutlich schneller, ein Element an beliebiger Stelle einzufügen oder zu löschen (lediglich Referenzen der Vorgänger und Nachfolger müssen angepasst werden)

# LinkedList

## Bp. für Methoden der Schnittstellen Collection und List

```
import java.util.LinkedList;
import java.util.List;

public class Warenkorb {

    private float artikelSumme;
    private List<Artikel> artikelliste = new LinkedList<Artikel>();
    ...

    public boolean artikelHinzufuegen(int position, Artikel a){
        try {
            artikelliste.add(position, a);
            artikelSumme += a.getPreis();
        } catch (IndexOutOfBoundsException ex) {
            return false; // Position ist ungültig
        }
        return true;
    }
}
```

Die benötigten Klassen aus dem Paket java.util müssen importiert werden

Hier wird die Implementierung einmalig definiert

Einsatzbeispiel für Methoden der Schnittstelle java.util.List

22.12.2016

Monika Tepfenhart

37

# LinkedList

## Bp. für Methoden der Schnittstellen Collection und List

```
public boolean artikelHinzufuegen(Artikel a){
    boolean erfolgreich = artikelliste.add(a);
    if (erfolgreich)
        artikelSumme += a.getPreis();
    return erfolgreich;
}

public boolean artikelEntfernen(Artikel a){
    boolean erfolgreich = artikelliste.remove(a);
    if (erfolgreich)
        artikelSumme -= a.getPreis();
    return erfolgreich;
}

public void leereWarenkorb(){
    artikelliste.clear();
    artikelSumme = 0;
}

public int getAnzahlArtikel(){
    return artikelliste.size();
}
```

Einsatzbeispiele für die Methoden der Schnittstelle java.util.Collection

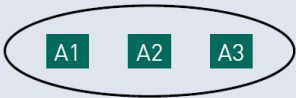
22.12.2016

Monika Tepfenhart

38

Mengen-Datenstrukturen

- Verwaltung zusammengehöriger Elementen; Reihenfolge keine Rolle; doppelte Elemente dürfen nicht vorkommen

Skizze	Interfaces/Klassen im Collections-Framework
	<p>Interface <code>java.util.Set</code> Beispiele für Implementierungen:</p> <ul style="list-style-type: none"><li>• <code>java.util.TreeSet</code> (als Baum realisiert)</li><li>• <code>java.util.HashSet</code> (als Hash-Tabelle)</li></ul>

Mengen-Datenstrukturen

- Implementierungen dieses Interfaces stellen sicher, dass kein Element doppelt vorhanden ist
- Es wird sichergestellt, dass folgende Bedingung nie für ein beliebiges Element-Paar `x` und `y` gilt: `x.equals(y) == true`
- Nur Methoden des Collections-Interfaces

## TreeSet

---

### ➤ TreeSet

- verwaltet Elemente der Menge intern in einer Baum-Struktur
- Operationen wie add, remove, contains können selbst bei sehr großen Datenmengen noch mit vertretbarem Aufwand durchgeführt werden

## HashSet

---

### ➤ HashSet

- verwaltet Menge mithilfe einer Hash-Tabelle
- konstanter Aufwand bei Operationen sichergestellt (vorausgesetzt die Hash-Funktion besitzt eine gute Streuung)
- Geschwindigkeit beim lesenden Zugriff auf die Elemente steht einem etwas höheren Aufwand beim schreibenden Zugriff gegenüber, da die Kosten zur Berechnung der Hash-Funktion berücksichtigt werden müssen.